Álgebra Linear Computacional - SVD e o algoritmo QR

Leonardo Herdy Marinho

[1]Programa de Pós Graduação em Informática – Universidade Federal do Rio de Janeiro (UFRJ)
Rio de Janeiro – RJ – Brasil

leonardomarinho@ufrj.br

Resumo. *Este trabalho final visa detalhar o que é o Algoritmo QR, sua utilidade prática e demonstrar exemplos de como podemos utilizá-lo. Também é intuito deste trabalho revisar conceitos sobre matrizes, decomposição de matrizes, o que um SVD e afins.*

Trabalho para a disciplina de álgebra linear computacional cursada em 2020 e aplicada pelo professor João A. R. Paixão no PPGI - UFRJ. Trabalho entregue em Novembro de 2020

1. Aprendizado agregado neste trabalho

Antes de iniciar o conteúdo do trabalho, é válido ressaltar todo o aprendizado que foi agregado neste trabalho. Com ele, consegui conhecer o algoritmo QR de maneira história e sobre sua aplicação. Revisei conceitos sobre a utilidade e para o que pode ser aplicado um SVD. Também revisei conceitos sobre matrizes (que era o único conhecimento que eu tinha previamente e que aprendi na escola antes de entrar na disciplina), consegui praticar um pouco mais o uso da linguagem Julia, aprendi a plotar matrizes com esquema de cores para entender melhor as modificações geradas a cada etapa do trabalho, tive a oportunidade de estudar um pouco mais sobre a decomposição QR e entender melhor a sua aplicação, notei que havia um erro estranho na visualização da matriz triangular superior após a aplicação do algoritmo QR então fui pesquisar como poderia refinar o resultado e aplicar uma tolerância para melhorar a matriz e consequentemente sua visualização e de bônus consegui criar um pequeno algoritmo com loops para substituir valores mediante uma tolerância de aproximação, consegui validar o correto funcionamento do algoritmo através das representações gráficas e ainda pratiquei inglês! :D

2. Matrizes

Matrizes são elementos matemáticos que estão ao nosso redor todos os dias, o que nem sempre paramos para perceber. As telas de nossos dispositivos eletrônicos (celulares, televisões, tablets, computadores, telões de LED e etc) foram criados utilizando a ideia de matriz, a compressão de arquivos como imagens, vídeos, áudios e documentos também só é possível através da organização dos dados de maneira que com cálculos matriciais conseguimos fazer esta engenhosa ”mágica”acontecer. Toda a nossa comunicação de voz e vídeo pela internet como conhecemos hoje foi viável graças ao uso de matrizes. Elas podem ser somadas, multiplicadas, subtraídas, unidas, separadas e invertidas de várias maneiras e as utilizamos para resolver sistemas lineares e não lineares. Também podemos utilizar as matrizes para se comportarem como verdadeiras funções matemáticas, assim, alterando valores de acordo com a necessidade do problema em que estamos trabalhando.

Uma boa ótica é enxergar as matrizes como uma espécie de ”massa de modelar”da qual podemos realizar diversas operações de maneira que seja possível obter o resultado esperado. Existem alguns tipos de matrizes, sendo eles:

- Matriz quadrada: Uma matriz recebe este nome quando o número de linhas e colunas (m,n) são iguais. Uma matriz de n = 8 e m = 8 terá 8 linhas e 8 colunas.
- Matriz diagonal: É aquela quando todos os elementos acima e abaixo da diagonal principal são nulos (iguais à zero).
- Matriz triangular: É aquela quando todos os elementos acima ou abaixo da diagonal principal são nulos (iguais à zero).
- Matriz identidade: É aquela onde todos os elementos da diagonal principal é igual a 1 e todos os outros elementos acima e abaixo da diagonal principal são nulos.
- Matriz nula: Uma matriz recebe o nome de nula quando todos os elementos, inclusive os presentes na diagonal principal são iguais a zero.
- Matriz linha: É aquele tipo de matriz que existe apenas uma linha. Ou seja, ”m”será sempre igual a 1.
- Matriz coluna: É aquele tipo de matriz onde existe apenas uma coluna. Ou seja, “n” será sempre igual a 1.

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{i1} & a_{i2} & \cdots & a_{in} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}$$

Figura 1. Exemplo de uma matriz

3. Decomposição de matrizes

Matrizes podem ser decompostas de diversas maneiras, aqui podemos listar algumas formas de decomposições como: decomposição LU, diagonalização de matrizes, decomposição simétrica e anti- simétrica, decomposição QR, decomposição Schur (mais utilizada para cálculos na área econômica) e decomposição em valores singulares (SVD). O tipo de decomposição que será escolhida depende do tipo de problema a ser resolvido, nível de complexidade deste problema e "custo"que esperam pagar computacionalmente para resolver um problema ao utilizar computadores para calcularem a decomposição. Algumas vezes, existe solução para um problema, mas, o tamanho da carga computacional é tão expressivo que é necessário estipular uma margem de erro aceitável que torne viável o uso da abordagem já que em muitos casos não haverá uma resposta precisa ao problema.

4. SVD, é de comer?

O termo SVD vem da língua inglesa que significa *Singular Value Decomposition*, ou em tradução livre, decomposição em valores singulares. A decomposição em valores singulares é considerado um dos resultados mais importantes obtidos em álgebra linear, tanto computacional quanto teoricamente falando. Assim como a decomposição LU ou QR, o SVD também é um tipo de decomposição e dentre todas é a mais confiável. Porém, seu uso tem prós e contras. De início posso citar o contra principal da utilização para o SVD que é o custo computacional. É bastante custoso executar cálculos com ele, sendo assim, é uma opção bastante interessante porém deve ser estudada com cautela ao ser escolhida para a execução de algum procedimento já que implica em custos de processamento o que financeiramente é bastante caro quando colocamos em pauta uma arquitetura de infraestrutura em nuvem, por exemplo. O SVD deve ser enxergado como a "bazuca"da álgebra linear, ou seja, resolve um oceano de problemas mas faz um enorme estrago se mal utilizado.

O SVD é a base dos métodos mais precisos para a solução de problemas de mínimos quadrados, para determinar o posto de matrizes, do espaço-imagem e do espaço nulo de matrizes, e da solução de vários problemas envolvendo normas euclidianas. Ele decompõe uma determinada matriz em um produto dos fatores de outras três matrizes:

$$A = USV$$

Considerando que U e V são matrizes ortogonais e S é diagonal. O número de valores singulares diferentes de zero na matriz é igual ao rank (posto) da matriz. Os valores da matriz diagonal são chamados de valores singulares, por isso a decomposição recebe este nome. Diversas linguagens como Python, Julia, Matlab, javascript, C, C++ e afins contém bibliotecas de álgebra linear que implementam a função SVD que recebe uma matriz de valores e retorna a decomposição. A Figura 2 ilustra um exemplo de decomposição de matriz utilizando svd realizada na linguagem Julia através da biblioteca LinearAlgebra.

Respondendo a pergunta do título, não, SVD não é de comer. Mas, se não tomar cuidado ele é bem guloso e pode comer toda a capacidade de processamento de uma máquina.

```
In [2]: using LinearAlgebra

        A = [1 2 2; 2 4 7;9 8 7]

        svd(A)

Out[2]: SVD{Float64,Float64,Array{Float64,2}}
        U factor:
        3×3 Array{Float64,2}:
         -0.181382  -0.147268  -0.972323
         -0.476212  -0.851913   0.217866
         -0.86042    0.502549   0.0843914
        singular values:
        3-element Array{Float64,1}:
         16.031670816527527
          3.8403246569385048
          0.4872754455233449
        Vt factor:
        3×3 Array{Float64,2}:
         -0.553753  -0.570806   -0.606249
          0.695735   0.0828584  -0.713504
          0.457505  -0.816894    0.351246
```

Figura 2. Exemplo de uma decomposição de matriz com SVD

5. Algumas aplicação prática do SVD

O SVD como já citado anteriormente é uma grande "bazuca"solucionadora de problemas algébricos. Existem inúmeras formas de aplicar SVDs para melhorar o nosso cotidiano como sociedade. Não cabe aqui espaço para detalhar todas as aplicações de um SVD, mas, podemos dar algumas como exemplo:

- Calcular a pseudo-inversa de uma matriz: O SVD é utilizado para calcular a pseudo-inversa de uma matriz. Um uso comum da pseudo-inversa é calcular uma solução que melhor se ajuste para um sistema linear que necessita de solução. Outro uso é encontrar a solução de norma mínima (euclidiana) para um sistema de equações lineares com diversas soluções. O cálculo da pseudo-inversa facilita a declaração e prova dos resultados em álgebra linear.
- Análise de Componentes Principais (PCA): A Análise de Componentes Principais, ou em inglês, PCA (Principal Component Analysis) é uma técnica amplamente utilizada de análise multivariada. PCA é vista como uma transformação linear ortogonal que transforma os dados para um novo sistema de coordenadas de forma que a maior variância por qualquer projeção dos dados fica ao longo da primeira coordenada (o chamado primeiro componente), a segunda maior variância fica ao longo da segunda coordenada, e assim por diante.
- Redução de dimensão e compressão de dados: Utilizamos o SVD para a compressão de dados (sejam eles áudios, vídeos, documentos ou imagens). Os casos normalmente mais encontrados de exemplos são as imagens por ser bastante simples de visualmente observar a mudança na qualidade conforme o número do rank (posto) da matriz que compõe a imagem sobe. A ideia é obter o maior teor de compressão com o menor número possível de perda de qualidade, ou seja, criar uma imagem visualmente aceitável com o mínimo possível de vetores na matriz

que a constrói. Nem sempre é uma tarefa trivial, mas, é completamente possível e o SVD é uma forma de realizar esta tarefa. Matematicamente falando, imagens são matrizes NxM. O valor máximo que um pixel f(i,j) da matriz pode atingir varia até f(i, j) = [0, 255]. A Figura 3 demonstra um exemplo de compressão de imagem sendo realizada através da aplicação do SVD sobre a matriz construtora da imagem.

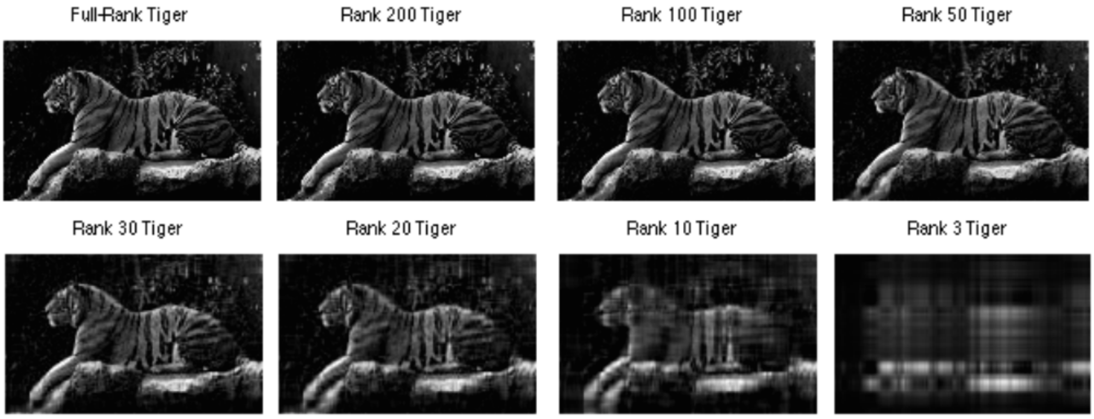

Figura 3. Exemplo de compressão de imagem com SVD

Considerando que *full rank* é a imagem original, quando o SVD chegou em rank 50 (posto) visualmente falando obtivemos uma imagem com uma qualidade aceitável e bastante próxima da original. As diferenças são praticamente imperceptíveis ao olho humano. O SVD assim, consegue reduzir o posto da matriz que compõe os dados necessários para a formação da imagem. Com menos postos precisamos de menos dados. Com menos dados, menos espaço é consumido para fazer o mesmo que era feito antes da aplicação do SVD. Obviamente existem perdas na qualidade da imagem, mas, é possível encontrar um número de postos que ao olho humano acaba sendo praticamente imperceptível a diferença entre uma imagem original e uma imagem comprimida através do SVD. O "santo graal"da compressão é um baixo número de postos e qualidade convincente que torna a operação um custo benefício satisfatório.

5.1. Algoritmo QR

O algoritmo QR, ou, também conhecido como iteração QR é um algoritmo de autovalor. Ele é utilizado para calcular os autovalores e autovetores de uma matriz. Apesar de bastante utilizado atualmente este algoritmo nasceu no final da década de 1950, seus criadores foram John GF Francis e Vera N. Kublanovskaya. A ideia que embasou a criação do Algoritmo QR foi a ideia básica de realizar uma decomposição QR escrevendo a matriz como um produto de uma matriz ortogonal e uma matriz triangular superior, multiplicar os fatores na ordem reversa e iterar.

O algoritmo QR foi precedido pelo algoritmo LR que usa a decomposição LU em vez da decomposição QR. O algoritmo QR é mais estável então o algoritmo LR raramente é usado atualmente. Porém, representa uma etapa importante para o desenvolvimento do algoritmo QR. O algoritmo LR foi desenvolvido no início dos anos 1950 por Heinz

Rutishauser , que trabalhava na época como assistente de pesquisa de Eduard Stiefel na ETH Zurich.

Bom, depois de um pouco de história que tal falarmos um pouco mais da parte técnica? O trabalho realizado pelo algoritmo QR em cima de uma matriz pode ser visto na Figura 4, bem interessante não é? O mais legal é a baixa complexidade a nível de código, com pouquíssimas linhas é possível criá-lo. Mas, como essa ”mágica”é feita?

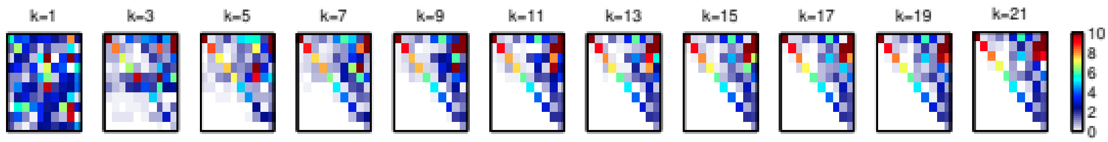

Figura 4. Resultado da utilização do algoritmo QR

O algoritmo QR contém três variantes, sendo elas:

- Calcular os valores próprios (autovalores) para uma matriz simétrica
- Calcular os valores próprios (autovalores) de uma matriz não simétrica
- Calcular os valores singulares de uma matriz retangular

Matematicamente falando, este algoritmo é um dos mais utilizados para o cálculo de autovalores e seus autovetores associados em matrizes. A discussão que desencadeou o surgimento do algoritmo vem de duas equações. A Figura 5.

$$A_{k-1} = Q_k Rk, \qquad R_k Q_k = A_k.$$

Figura 5. Equações que originaram o algoritmo QR

Podemos traduzir estas equações de maneira algorítmica da seguinte forma: A partir das iterações k-1 realizadas por um loop, faça uma fatoração QR da matriz A e atribua para a mesma a multiplicação de R e Q para ela. Com isso, a cada iteração do loop k a fatoração QR é realizada em cima da matriz que foi fatorada anteriormente.

Traduzindo para algoritmos, podemos entender estas funções como:

Algorithm 1: Algoritmo QR simples, ou, Iteração QR

Result: Uma matriz triangular superior
Inicialização;
for *k = 1, 2, ...* **do**
    Calcule a decomposição QR de A;
    Atribua o produto de R e Q à A.
end

O resultado esperado da aplicação deste algoritmo é uma matriz como desenhada na Figura 6.

Agora conseguimos entender melhor o motivo do algoritmo QR também ser conhecido como iteração QR. A ideia é que haja uma ”eliminação dos bojos”a cada iteração realizada até que o resultado se torne uma matriz

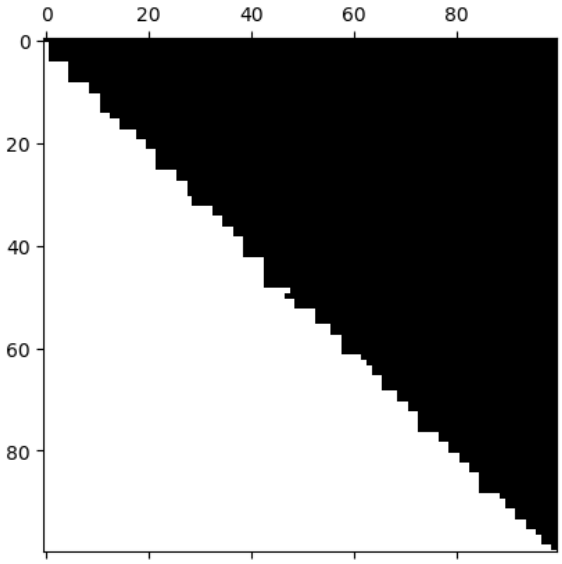

Figura 6. Resultado esperado

A Figura 7 demonstra o passo-a-passo das transformações, fazendo com que o bojo, vá para a parte de baixo.

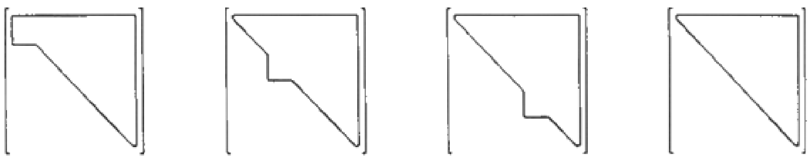

Figura 7. Esquema de eliminação de bojos

5.2. Criação e execução prática do algoritmo e autovalores não simétricos

Para exemplificar a aplicação do algoritmo QR utilizaremos a linguagem Julia por ser uma linguagem com uma boa legibilidade quando o assunto é álgebra linear.

Para iniciarmos o exemplo é necessário criar uma matriz que será chamada de A, optamos por uma matriz quadrada de dimensões 8x8 para este exemplo tendo em vista que é um tamanho computacionalmente falando razoável para realizarmos os cálculos e também tem informações suficientes para plotarmos em imagens e criar a demonstração visual de seu funcionamento em forma de imagem.

A Figura 8 demonstra como realizamos a criação da matriz A e quais são as bibliotecas que utilizaremos para a criação do exemplo assim como também informamos como instalar as bibliotecas *Colors* e *Plots*.

```
In [150]: # Executar na linha de comando antes de executar esse notebook
          # import Pkg; Pkg.add("Colors"); Pkg.add("Plots");

          using Colors, Plots

          A = randn(8,8)

Out[150]: 8×8 Array{Float64,2}:
            0.808615   0.571149    1.0961    …  -1.67668    1.11499   -0.57261
           -0.302238   1.63289     2.07384      -0.395042  -1.25368   -0.493676
           -1.23865   -0.377258    0.702535      0.282548  -0.239424   2.01458
            0.562315   0.38584     0.854391     -1.89319   -0.366691   0.139241
            0.943869  -0.0540654   0.830537     -0.528799   0.417801  -1.04335
            0.640392   0.201279   -0.29342   …  -0.455844   0.171824  -1.00299
            0.882473  -0.27037    -1.7112        1.73647   -0.237538   0.538596
            1.39702    1.04306     0.667123      0.711622   1.69027    0.278385
```

Figura 8. Criação da matriz A e imports

O comando *randn* criou para nós uma matriz quadrada de valores randômicos (aleatórios) com a dimensão de 8 linhas por 8 colunas. Feito isso podemos utilizar um comando da biblioteca *Plots* para gerar uma visualização gráfica da matriz e gerar um melhor entendimento do estado atual da matriz para acompanharmos posteriormente as alterações. A Figura 9 demonstra o estado atual da matriz.

```
In [151]: # Plotando a matriz A, podemos observar como ela se comporta
          heatmap(A, yflip=true)
```

Out[151]:

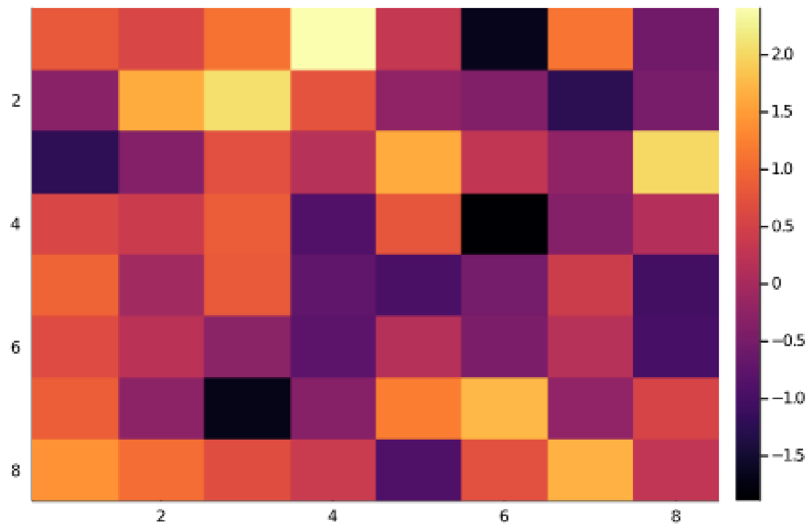

Figura 9. Plot da matriz matriz A

Criada a matriz A, implementaremos o algoritmo QR para trabalhar em cima dela e nos retornar o valor indicado. A Figura 10 demonstra em detalhes a implementação do algoritmo na linguagem Julia e mostra como ficou a matriz de saída (resultado). De cara conseguimos perceber que algo muito interessante aconteceu ao observarmos os zeros na parte inferior esquerda.

```
# Algoritmo qr
for k = 1:1000
    Q, R = qr(A)
    A = R*Q
end


show(stdout, "text/plain", A)
```

```
8×8 Array{Float64,2}:
 -3.07735       -1.44322       -0.697018  -2.22203    0.778562  -0.0816514     -0.335063      0.879193
  0.614234      -2.08185       -1.0195     1.01249   -0.979507  -0.27728       -0.489439     -0.286065
  5.99066e-89   -3.24872e-88    1.54265   -1.46421   -1.02673   -1.38053        0.151513     -0.429575
  3.16368e-90    6.49422e-89    0.492554   1.88522   -2.48045   -1.01201        1.39984       0.218197
  6.26005e-90   -6.56483e-89    1.29533    0.899344   1.17299    0.827597      -1.31166      -0.367287
 -2.0e-323       0.0            0.0        0.0       -2.0e-323   1.51868        0.85742      -1.38133
  0.0            0.0            0.0        0.0        0.0        3.55765e-290  -1.08839       0.632915
  0.0            0.0            0.0        0.0        0.0        1.0e-323      -3.30673e-88   0.984525
```

Figura 10. Algoritmo QR em ação

Rufem os tambores! Vamos plotar a matriz A após a aplicação do algoritmo e ver se de fato conseguimos o resultado esperado! A Figura 11 demonstra o resultado da aplicação do algoritmo.

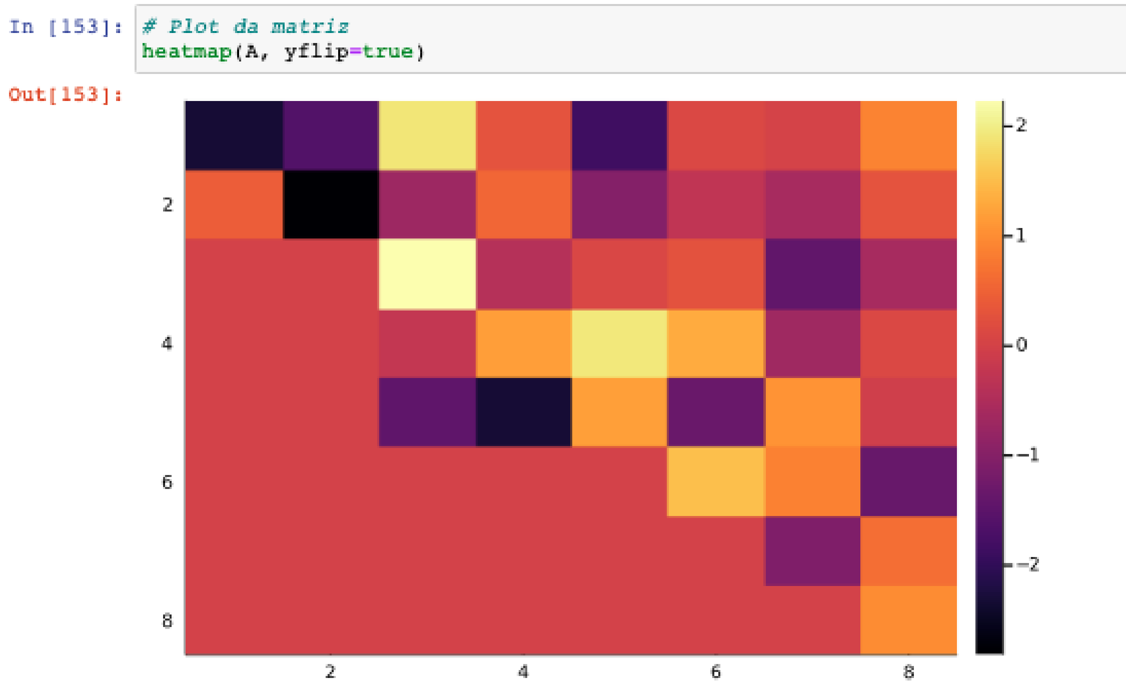

Figura 11. Algoritmo QR em ação

E, funcionou! Temos um algoritmo QR simples executado com sucesso em autovalores não simétricos! Vale também a observação de que a Figura 11 não trouxe uma matriz triangular perfeita, isso pode ser resolvido estipulando uma margem de erro que despreze na exibição visual da matriz valores muito pequenos como -2.0e-323, por exemplo. Para corrigir este problema, podemos criar em Julia um loop que verifique os valores da matriz e substitua-os por 0 caso sejam iguais ou menores a uma tolerância. A Figura 12 demonstra a aplicação da tolerância de aproximação na matriz.

```
# Aplica uma tolerância de erro (1e-10) tal que substitua valores menores que ela por 0

for linha = 1:size(A)[1]
    for coluna = 1:size(A)[2]
        if(abs(A[linha, coluna]) < 1e-10)
            A[linha, coluna] = 0.0
        end
    end
end

show(stdout, "text/plain", A)
```

```
8×8 Array{Float64,2}:
 -0.875585  -2.94385    0.568331   1.11525   -0.930865   1.70015   -1.17342     -1.22612
  2.82158   -0.660147  -0.171773  -0.194736   1.2764    -0.172479   0.35634      0.972663
  0.0        0.0       -2.245     -0.489908  -0.616511  -2.0595     0.00986161   0.349004
  0.0        0.0        0.0        1.0967    -1.79765    0.477342   1.77365      0.0399888
  0.0        0.0        0.0        0.256803   1.93814    0.575982  -0.952168    -0.942795
  0.0        0.0        0.0        0.0        0.0       -0.669051   1.60188      0.573924
  0.0        0.0        0.0        0.0        0.0       -1.13903    0.567397     0.179245
  0.0        0.0        0.0        0.0        0.0        0.0        0.0          0.172585
```

Figura 12. Aplicação da tolerância de aproximação na matriz A

Ao plotarmos conseguimos ver uma melhora na exibição da matriz já que valores que geravam ruídos na visualização foram removidos. Em uma matriz pequena como a que estamos utilizando 8x8 a diferença pode ser bem sutil, mas, em matrizes de escalas maiores a diferença da aplicação desta tolerância de aproximação é extremamente visível. A Figura 13 demonstra o Plot da matriz A após refinarmos seus valores com a aplicação da tolerância.

```
In [201]: # Plot da matriz
          heatmap(A, yflip=true)
```

Out[201]:

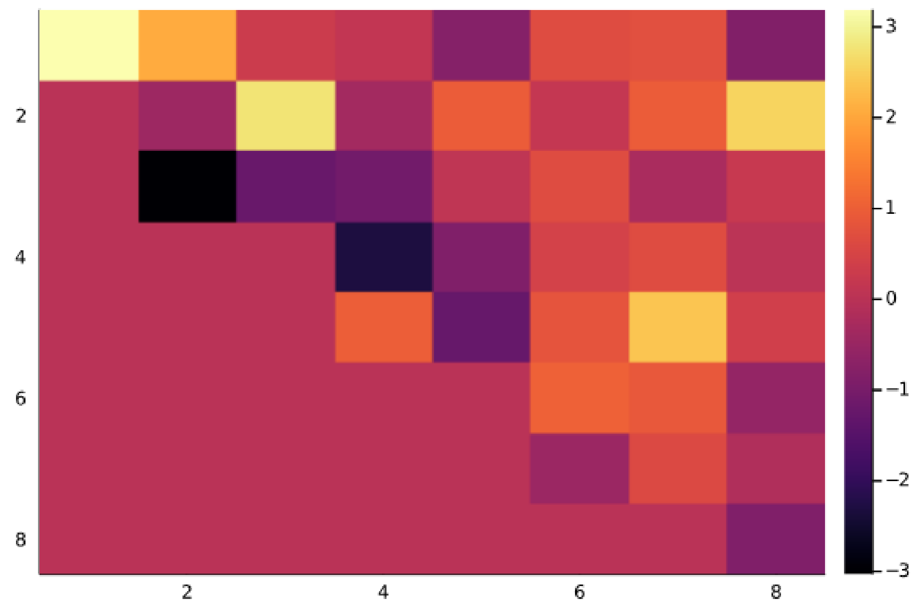

Figura 13. Plot da tolerância de aproximação aplicada na matriz A

Vale ressaltar que os números demonstrados nas matrizes representadas nas Figuras de exemplo deste trabalho variam devido as execuções sucessivas do código para a retirada dos prints, correções no trabalho e afins. Por termos utilizado uma função randômica para gerar a matriz A, a cada nova atualização os valores mudam, porém, os resultados apresentados neste trabalho conseguem ser obtidos de maneira praticamente igual quando o notebook é reexecutado.

5.3. Autovalores simétricos

Para criarmos a matriz simétrica podemos utilizar o seguinte código na linguagem Julia:

$$S = (A + A')/2$$

Feito isso, conseguimos criar uma matriz com base na matriz A que seja simétrica, ou seja, criaremos um efeito ”espelho”na matriz A de maneira que seja uma matriz com todos os seus valores atribuídos e que quando plotada possamos visualmente notar que existe uma diagonal central que divide a parte superior da parte inferior e que o conteúdo de ambas as partes é o mesmo. A Figura 14 demonstra este trabalho de tornar a matriz simétrica e a visualização dela.

```
# Trabalhando com Autovalores simétricos
A_simetrica = (A + A') / 2
```

```
8×8 Array{Float64,2}:
 -3.23773    -0.210307  -0.358793   …  -0.258681    0.0831983   -0.691248
 -0.210307    2.70193    0.818786       0.718193    0.102372     0.548922
 -0.358793    0.818786   1.56027       -1.05838    -0.402084    -1.77333
 -0.103577   -0.231349  -0.266017      -0.33364     0.418794     0.421647
 -0.252649   -0.552772  -0.0975194     -0.374743   -0.00665586   0.275907
 -0.258681    0.718193  -1.05838    …   0.371999   -0.194045     0.303788
  0.0831983   0.102372  -0.402084      -0.194045   -0.727144     0.144568
 -0.691248    0.548922  -1.77333        0.303788    0.144568     0.162818
```

Figura 14. Matriz S simétrica a matriz A

Podemos agora plotar esta matriz e observar que existe uma diagonal que começa no lado superior esquerdo e termina no lado inferior direito que divide as duas partes da matriz que visualmente observamos a simetria. A Figura 15 demonstra a matriz da Figura 14 de maneira visual.

In [185]:
```
# Visualizando o que mudou da A para A_Simetrica
heatmap(A_simetrica, yflip=true)
```
Out[185]:

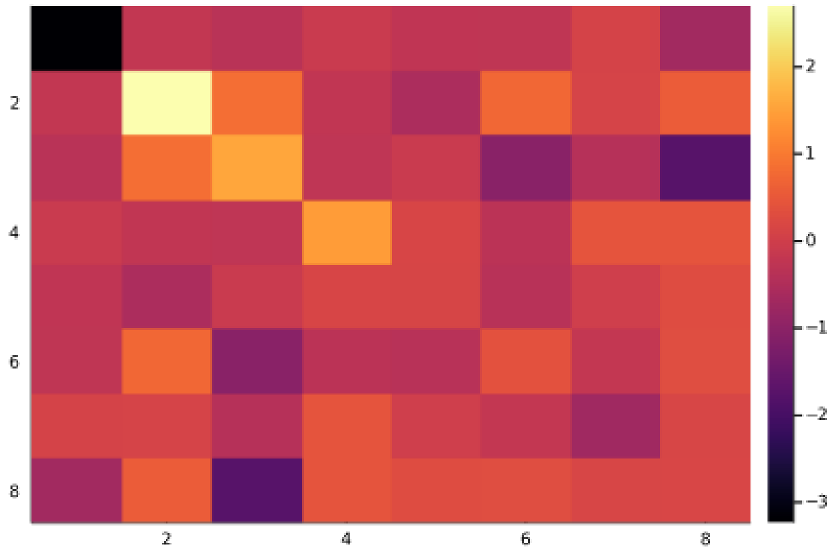

Figura 15. Plot da matriz simétrica

A Figura 16 demonstra melhor sem a diagonal principal a simetria entre as partes.

In [185]:
```
# Visualizando o que mudou da A para A_Simetrica
heatmap(A_simetrica, yflip=true)
```
Out[185]:

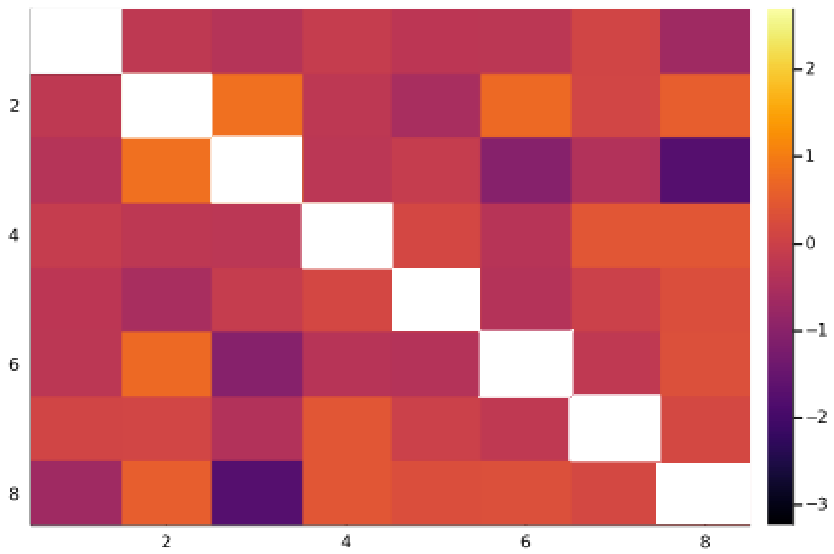

Figura 16. Plot da matriz simétrica sem a diagonal principal

O interessante é que se reaplicarmos o algoritmo QR na matriz porém com ela simétrica, através desta simetria não só as colunas mas também as linhas são zeradas de maneira que só sobre a diagonal principal da matriz. Com isso, obtemos uma matriz diagonal. A Figura 17 demonstra este experimento.

```
for k = 1:100
    Q, R = qr(A_simetrica)
    A_simetrica = R*Q
end

show(stdout, "text/plain", A_simetrica)
```

```
8×8 Array{Float64,2}:
 -3.53165        -0.000365183     6.32436e-13     2.7444e-17     -2.15042e-16    -2.21409e-16   -1.3271e-16     5.26076e-17
 -0.000365183     3.43337         2.19511e-10    -5.47317e-17    -6.53702e-16    -1.29982e-16    2.58713e-16    1.17406e-16
  6.3224e-13      2.19511e-10     3.08792        -8.88851e-17     1.02764e-16     3.30969e-16   -3.40976e-16    2.77138e-16
 -3.33352e-71    -2.92161e-68     2.47062e-60     1.58173        -1.68646e-14     4.00767e-16    3.29631e-16   -3.58586e-17
 -4.1248e-83      2.46285e-80    -7.55303e-71    -1.6979e-14     -1.36879         2.50313e-16    1.80623e-16   -1.48571e-16
 -2.90145e-130    1.8734e-127    -5.95027e-118    7.85934e-62    -5.12861e-48    -0.793124       2.14142e-16   -1.70172e-16
  8.54817e-209    8.22247e-206   -3.78941e-196    3.02198e-138    1.05429e-125   -2.85104e-78   -0.324899       8.07404e-8
  2.83474e-214   -9.17715e-212    3.55387e-202   -2.51416e-145    1.9599e-131    -1.51395e-84    8.07404e-8     0.301091
```

Figura 17. Matriz diagonal

Eu disse que conseguimos uma matriz diagonal, mas, será que de fato conseguimos mesmo? É meio difícil identificar visualmente com tantos valores grandes. Mas, se aplicarmos nosso algoritmo que aplica uma tolerância de erro conseguimos obter um resultado visualmente melhor. A Figura 18 demonstra para nós de maneira visualmente melhor a diagonal da matriz. Ainda com alguns valores bem aproximados a 0 aparecendo e gerando algum ruído, mas, bem mais simples de identificar a diagonal.

```
# Aplica uma tolerância de erro (1e-10) tal que substitua valores menores que ela por 0

for linha = 1:size(A_simetrica)[1]
    for coluna = 1:size(A_simetrica)[2]
        if(abs(A_simetrica[linha, coluna]) < 1e-10)
            A_simetrica[linha, coluna] = 0.0
        end
    end
end

show(stdout, "text/plain", A_simetrica)
```

```
8×8 Array{Float64,2}:
 4.01893  0.0       0.0           0.0           0.0           0.0           0.0       0.0
 0.0      2.17471   0.0           0.0           0.0           0.0           0.0       0.0
 0.0      0.0       1.69799      -0.000520955   0.0           0.0           0.0       0.0
 0.0      0.0      -0.000520955  -1.55328       3.63486e-10   0.0           0.0       0.0
 0.0      0.0       0.0           3.63486e-10  -1.2749       -0.000987967   0.0       0.0
 0.0      0.0       0.0           0.0          -0.000987967  -1.24309       0.0       0.0
 0.0      0.0       0.0           0.0           0.0           0.0           0.339568  0.0
 0.0      0.0       0.0           0.0           0.0           0.0           0.0       0.146931
```

Figura 18. Matriz diagonal refinada

Repare que chamamos a matriz resultante de matriz diagonal, mas, os valores dela não são explicitamente 0. Porém, se repararmos melhor, eles são valores elevados a expoentes altíssimos o que nos arremete a considerar uma margem de erro o que traz estes valores para o 0. A Figura 19 demonstra visualmente o tamanho desprezível dos valores que não são os da diagonal principal da matriz.

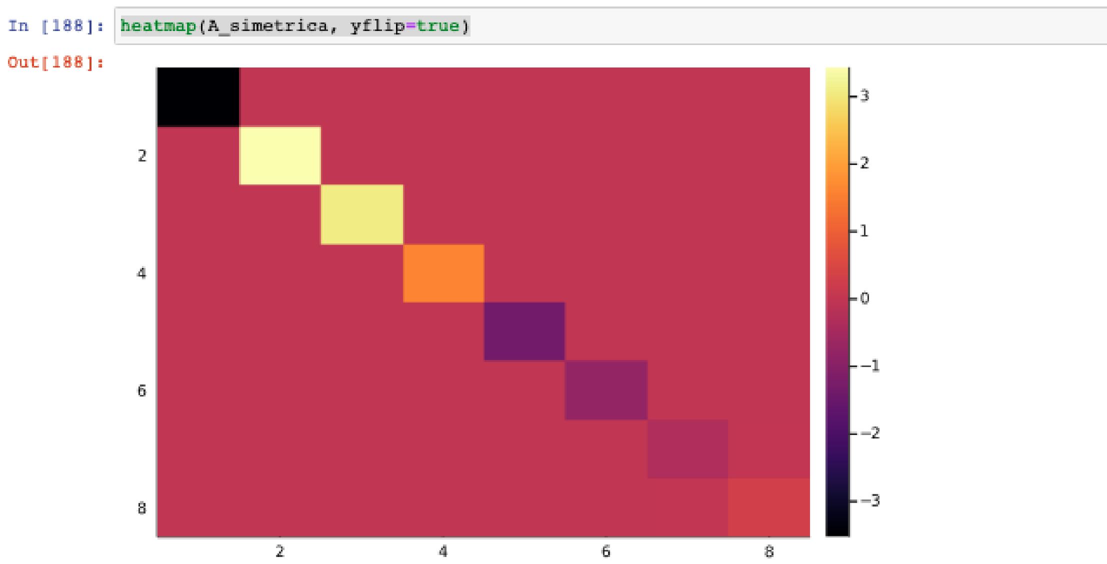

Figura 19. Matriz simétrica submetida ao algoritmo QR

5.4. SVD e algoritmo QR

A título de curiosidade, é possível aproximar os valores de maneira que mesmo que não estejamos em uma matriz quadrada seja possível encontrar a diagonal principal de maneira projetada (aproximada). Pelo SVD ser um algoritmo mais ”genérico”ele consegue lidar bem com matrizes retangulares e não só com a diagonalização de matrizes quadráticas. O SVD consiste em encontrar duas bases (em vez de apenas uma que é o caso de autovetores da diagonalização) que sejam adaptadas à matriz que no caso chamamos de A.

Para realizar o teste, podemos pegar a nossa matriz A quadrática (8, 8) e inserir duas linhas de uma matriz identidade de iguais dimensões. A Figura 20 demonstra como podemos realizar esta ação em Julia.

A Figura 21 demonstra a a plotagem da matriz que criamos.

```
matriz_identidade = Matrix{Float64}(I, 8, 8)
A = randn(Float64, (8,8))

show(stdout, "text/plain", matriz_identidade)

# Serve apenas para melhorar a visualização
println();println()

# Insere 2 linhas da matriz identidade para tornarmos a matriz A retangular
A = [A[1:4,:]; matriz_identidade[1,:]'; A[5:8,:]; matriz_identidade[2,:]']

show(stdout, "text/plain", A)
```

```
8×8 Array{Float64,2}:
 1.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  1.0  0.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  1.0  0.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  1.0  0.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  1.0  0.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  1.0  0.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  1.0  0.0
 0.0  0.0  0.0  0.0  0.0  0.0  0.0  1.0

10×8 Array{Float64,2}:
 -0.545912   0.205821   1.32041    0.497555   1.22233     0.224542   1.05746   -0.772542
 -1.67636    0.642832   1.28013    0.707878  -0.145686    0.682277   1.12804    0.206629
  0.285118   0.834936  -2.41239    1.92469    0.762382   -1.06272    1.58489   -0.230812
 -0.21946   -1.58242    0.385426  -0.34636    1.90766    -0.692602   1.16526   -2.18466
  1.0        0.0        0.0        0.0        0.0         0.0        0.0        0.0
  0.385296  -0.82469   -1.33192    0.258182   0.0613855  -0.325805   0.80932   -1.17248
  2.27145    0.228693  -1.52617   -0.440021  -0.377487    0.46617   -0.53994   -1.56349
 -0.764895   0.43433    1.29565   -0.705844  -0.361922   -0.769854   1.68267    0.572642
 -0.730455   2.26195    1.71027   -0.535825  -0.0817852   0.453964   0.263198  -1.14434
  0.0        1.0        0.0        0.0        0.0         0.0        0.0        0.0
```

Figura 20. Preparação da matriz para o uso de SVD

```
In [178]: heatmap(A, yflip=true)
```

Out[178]:

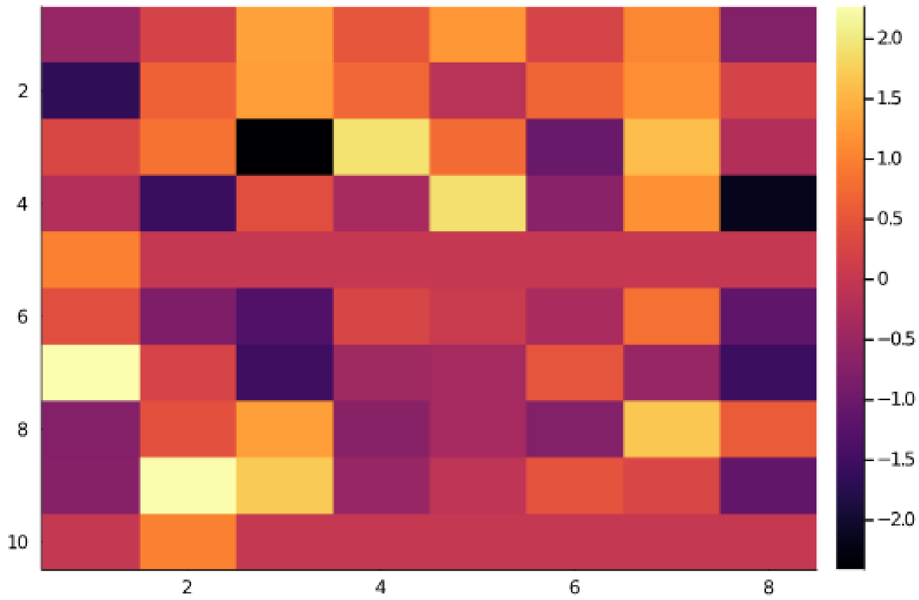

Figura 21. Preparação da matriz para o uso de SVD plotada

Utilizando uma decomposição QR Householder operando na matriz A da esquerda para zerar uma coluna e, em seguida, outro Householder operando da direita para zerar a maior parte de uma linha conseguimos criar uma diagonal ”bruta”, ou seja, com uma margem de erro como mostra a Figura 22.

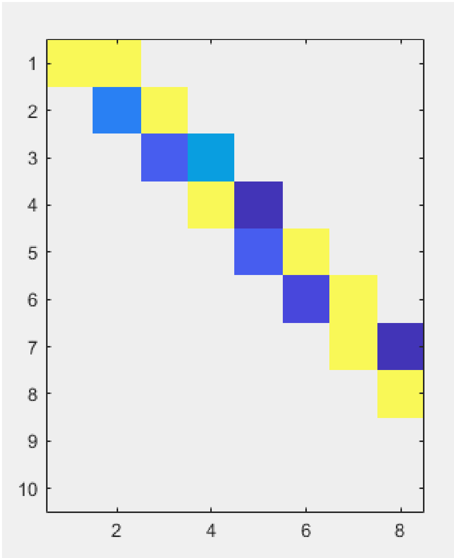

Figura 22. Diagonal

Após esta ação, caso apliquemos o algoritmo QR (ou iteração QR, como preferir) conseguimos reduzir o tamanho fora da diagonal de maneira significante. O que nos traz a diagonal que anteriormente foi gerada pelo algoritmo QR na matriz antes de inserirmos as linhas da matriz identidade. A Figura 23 mostra o resultado final desta operação.

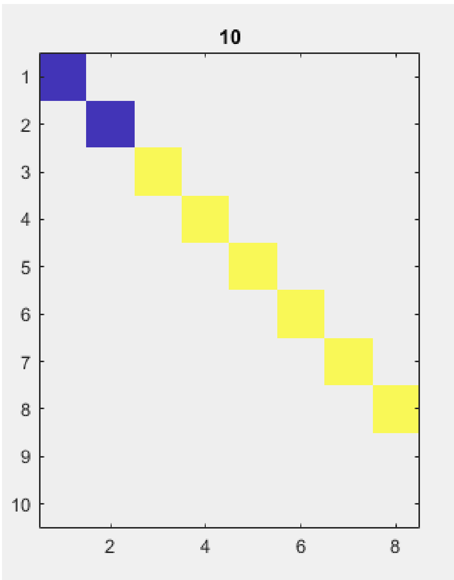

Figura 23. Diagonal

6. Conclusão

Com este trabalho foi possível entender melhor os conceitos sobre manipulação de matrizes (decomposição, varredura de matrizes e afins). Com a representação gráfica das matrizes geradas foi possível testar e validar se os cálculos foram realizados corretamente e se o resultado estava dentro do que era esperado. Todos os códigos e exemplos estão disponíveis através do seguinte endereço: `https://bit.ly/qr_algoritmo` em formado de Jupyter notebook ou arquivo Julia. É válido ressaltar que antes de executar os exemplos é necessário realizar a instalação das bibliotecas Colors e Plots.

Deixo aqui os meus agradecimentos pela oportunidade de aprender um pouco mais sobre matemática e perder o medo/preconceito com o uso da matemática na computação. Tenho certeza que ao longo da caminhada como mestre e programador ter conhecimentos matemáticas farão a diferença positivamente.

7. Referências bibliográficas

Para este trabalho buscamos conhecimento em várias vias de pesquisa, sendo elas:

http://www2.ime.unicamp.br//~ms512/sites/default/files/mlh.pdf
http://www2.ime.unicamp.br//~ms512/sites/default/files/lcp.pdf
https://web.stanford.edu/class/cme335/lecture6.pdf
http://www.math.iit.edu/~fass/477577_Chapter_12.pdf
https://sca.profmat-sbm.org.br/sca_v2/get_tcc3.php?id=45407
https://www.ufsj.edu.br/portal2-repositorio/File/nepomuceno/mn/08MN_SL6.pdf
http://www2.ime.unicamp.br//~ms512/sites/default/files/lcp.pdf
https://en.wikipedia.org/wiki/QR_algorithm
https://www.sciencedirect.com/science/article/pii/0024379593904798
http://www.math.iit.edu/~fass/477577_Chapter_12.pdf
https://core.ac.uk/download/pdf/82413528.pdf https://web.stanford.edu/class/cme335/lecture6.pdf
https://www.youtube.com/watch?v=_vlE7aJGoNE
https://docs.julialang.org/en/v1/